На этом занятии мы с вами рассмотрим **одну из ключевых возможностей ООП языка C++ – чисто виртуальные методы (функции) и абстрактные классы**. Благодаря этим конструкциям и механизму наследования появляется возможность описывать логику программы на обобщенном уровне, не привязываясь к конкретной реализации. И сейчас мы с вами начнем знакомиться с магией такого подхода, дарованного объектно-ориентированным программированием.

На предыдущем занятии были описаны два класса: базовый GeomBase и дочерний Line:

```cpp
class GeomBase {
protected:
    int x0{0}, y0{0}, x1{0}, y1{0};
public:
    void set_coords(int x0, int y0, int x1, int y1)
    {
        this->x0 = x0; this->y0 = y0;
        this->x1 = x1; this->y1 = y1;
    }

    virtual void draw() const
    {
        printf("GeomBase: %d, %d, %d, %d\n", x0, y0, x1, y1);
    }
};

class Line : public GeomBase {
private:
    double length{0.0};
public:
    virtual void draw() const override
    {
        printf("Line: %d, %d, %d, %d\n", x0, y0, x1, y1);
    }
};
```

В базовом классе мы разместили виртуальный метод draw, который переопределили, затем, в дочернем классе Line. И было бы логично метод draw в базовом классе объявить без какой-либо реализации, т.к. рисование неизвестной фигуры занятие крайне странное. Но может нам тогда совсем

убрать этот метод из базового класса? Если мы так сделаем, то следующая конструкция завершится ошибкой:

```language-cpp
int main()
{
    Line* ptr_ln = new Line;
    GeomBase* ptr_b = ptr_ln;

    ptr_b->draw(); // ошибка, нет метода draw в классе GeomBase

    delete ptr_ln;
    delete ptr_b;
    return 0;
}
```

Чтобы это работало, метод draw должен с одной стороны присутствовать в классе GeomBase, но с другой – не иметь никакой реализации (тела). А вызываться будет аналогичный метод дочернего класса. Может нам тогда его записать в виде прототипа:

```language-cpp
virtual void draw() const;
```

Но и это не решение, т.к. прототип требует последующего определения тела метода, как правило, вне класса. В таком виде мы снова получим ошибку при его вызове.

Конечно, можно сделать своеобразный костыль и прописать пустое тело метода:

```language-cpp
virtual void draw() const { }
```

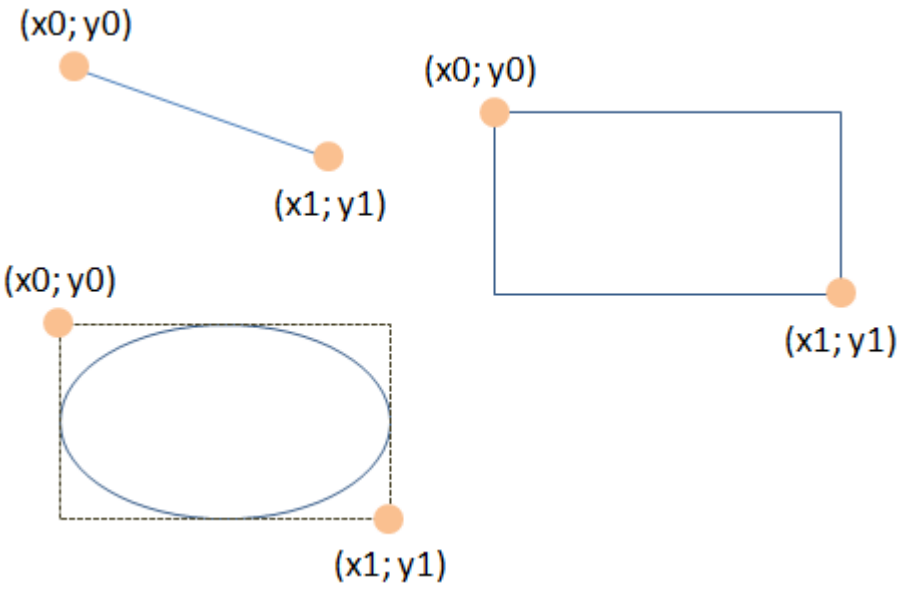
Но тогда в случае отсутствия этого метода в дочернем классе вызовется метод draw с пустым телом. А нам хотелось бы так объявить виртуальный метод draw, чтобы он гарантированно брался из дочернего класса, а вызов из базового был бы невозможен. Тогда программист случайно не совершит ошибку, если какой-либо важный метод забудет переопределить.

Так вот, существует еще один способ объявления виртуальных методов с отсутствующим телом, который предложил автор языка Бьерн Страуструп:

```language-cpp
virtual void draw() const = 0;
```

Здесь мы говорим компилятору, что **в базовом классе объявлена сигнатура виртуального метода draw (без реализации), который должен быть переопределен в дочернем классе**. Затем, используя этот прототип и указатель на базовый класс, **по таблице виртуальных функций дочернего класса следует брать метод аналогичный метод уже с реализацией**. То есть, определяя метод без тела, мы в базовом классе создам универсальный интерфейс для вызова произвольных методов с аналогичной сигнатурой из дочерних классов. Сами же виртуальные методы без реализаций получили название **чисто виртуальных методов (pure virtual function, method**)**.

Хорошо, но зачем нам все это понадобилось? Я напомню, что мы начинали тему наследования с примера описания различных графических примитивов:



Геометрия каждого примитива определяется двумя координатами на плоскости (четырьмя переменными: x0, y0, x1, y1). Давайте добавим еще два дочерних класса для прямоугольников и эллипсов:

```cpp
class Rect : public GeomBase {
public:
    virtual void draw() const override
        { printf("Rect: %d, %d, %d, %d\n", x0, y0, x1, y1); }
};

class Ellipse : public GeomBase {
public:
    virtual void draw() const override
        { printf("Ellipse: %d, %d, %d, %d\n", x0, y0, x1, y1); }
};
```

И, так как у всех у них единый базовый класс, то мы можем на программном уровне хранить эти объекты следующим образом:

```language-cpp
int main()
{
    GeomBase* g1 = new Line;
    GeomBase* g2 = new Rect;
    GeomBase* g3 = new Line;
    GeomBase* g4 = new Ellipse;

    delete g1;
    delete g2;
    delete g3;
    delete g4;

    return 0;
}
```

Или, сделать еще лучше, воспользоваться массивом указателей, например, так:

```language-cpp
enum {max_geoms = 1000};

int main()
{
    size_t count_g = 4; // число геометрических фигур
    GeomBase* geoms[max_geoms] = {nullptr};

    geoms[0] = new Line;
    geoms[1] = new Rect;
    geoms[2] = new Line;
    geoms[3] = new Ellipse;

    for(size_t i = 0;i < count_g; ++i)
        geoms[i]->draw();

    for(size_t i = 0;i < count_g; ++i)
        delete geoms[i];

    return 0;
}
```

Смотрите, используя указатели на базовый класс каждого примитива, мы на общем уровне прописали их отображение, используя чисто виртуальный

метод draw, который изначально (в базовом классе) не имел никакой реализации. По сути, фрагмент программы:

```cpp
    for(size_t i = 0;i < count_g; ++i)
        geoms[i]->draw();
```

никак не связан с конкретной реализацией отображения графических примитивов. Конкретика появляется только в дочерних классах. А раз так, то в будущем совершенно спокойно в эту программу можно добавить любой другой новый дочерний класс для нового примитива. И он автоматически встроится в работу программного кода, описанного на уровне виртуальных функций базового класса. Например, если прописать класс для представления кругов:

```cpp
class Circle : public GeomBase {
public:
    virtual void draw() const override
        { printf("Circle: %d, %d, %d, %d\n", x0, y0, x1, y1); }
};
```

То в функции main он достаточно просто встраивается в работу общей логики программы:

```cpp
int main()
{
    size_t count_g = 5; // число геометрических фигур
    GeomBase* geoms[max_geoms] = {nullptr};

    geoms[0] = new Line;
    geoms[1] = new Rect;
    geoms[2] = new Line;
    geoms[3] = new Ellipse;
    geoms[4] = new Circle;

    for(size_t i = 0;i < count_g; ++i)
        geoms[i]->draw();

    for(size_t i = 0;i < count_g; ++i)
        delete geoms[i];

    return 0;
}
```

Все, что нам нужно было сделать – это добавить новый объект класса Circle в массив geoms. После этого он будет обрабатываться наряду с другими,

ранее существующими объектами. Видите, как легко и просто можно расширять функционал программы, просто добавляя новые дочерние классы. И все благодаря использованию чисто виртуальных функций и механизму наследованию. А вызов в цикле метода draw:

```language-cpp
    for(size_t i = 0;i < count_g; ++i)
        geoms[i]->draw();
```

это пример **динамического полиморфизма**, когда вызываемый метод (draw) определяется не в момент компиляции программы, а в момент ее работы.

Абстрактные классы

В заключение этого занятия несколько замечаний о классе GeomBase, который стал содержать один чисто виртуальный метод. Наличие такого метода превращает класс GeomBase в **абстрактный класс**. Что это значит? Это значит, что объекты этого класса создавать не получится. Следующая команда приведет к ошибке на этапе компиляции:

```language-cpp
GeomBase* p = new GeomBase;
```

С чем это связано? Да, в классе имеется чисто виртуальный метод draw, без реализации и это переводит класс на такой уровень абстракции, где существование объектов становится уже невозможным. Поэтому **любой класс в языке C++, который содержит или наследует без переопределения хотя бы один чисто виртуальный метод, является абстрактным**.
Абстрактные классы, как мы уже видели из примера геометрических фигур, можно использовать при наследовании, создавая на их основе другие полноценные дочерние классы. Объекты дочерних классов можно спокойно создавать (при переопределении в них чистых виртуальных методов).